

Chapter 11

Pointers

11.1 L values and R values

11.1.1 R values

- R value is the right-hand side of an assignment.
 - `lvalue = rvalue`
- R values are numbers, constants or intermediate results of calculations
 - `4`, `(i + 5)`, ...
 - Not necessarily stored in memory

11.1.2 L values

- L value is the left-hand side of an assignment
 - Is always stored in memory
 - Variables are labelled memory segments, thus they are L values
 - Intermediate results of R values are not necessarily stored in memory
- *Every L value has a memory address*

11.1.3 Address operator

- Address operator `&` returns the address of a variable
 - Address is also often called reference
 - Placeholder for `printf`, `%p`, prints number as hexadecimal.
- *Address operator can only be applied to L values!*

11.2 Pointer variables

11.2.1 Variables that contain addresses

- Addresses are numbers and can be stored in variables
 - Size of an address is system dependant (usually 32 bit or 64 bit)
- Variables that contain addresses are called *pointers* or *references*.
 - Pointer variables contain the address of variables.
 - Thus, they can be used to send values to this address.

11.2.2 Real life analogy

- Ms Jones lives in `main road 16`.
 - Ms Jones is a variable.
- Someone writes the address `main road 16` on a letter.
 - The letter now is a *reference to* Ms Jones.
- Someone can now go to the address written on the letter and will find Ms Jones.
 - This is called *dereferencing*.

11.2.3 Dereference Operator

- `&` returns the address/reference of a variable
- `*` is its counterpart that returns *the value at a given address*.
 - The result of `*(main_road 16)` would be Ms Jones.

11.2.4 Dereferencing

- Pointer variables have the dereference operator as a prefix
- Assume, Ms Jones is an integer.
- Syntax: `int *letter`
 - Read as: By dereferencing `(*) letter` one obtains an integer (`int`).

```
int msjones = 65;
int *letter = &msjones; // letter contains the address
                        // of msjones.
printf("%d\n", *letter); // value at the given address
```

11.2.5 Example

- Two pointers can point to the same variable

```
int main() {
    int i;
    int *p = &i;
    int *q = &i;

    printf("%p %p\n", p, q);
}
```

11.3 Dereference Operator

11.3.1 Dereferencing a pointer

- Operator for dereferencing variables is `*`
 - Opposite of `&`.
 - `int *p`: If one applies the dereference operator, one obtains an integer.
 - (Keep in mind that `&` can only be applied to an L value)

11.3.2 Example 1

```
int main() {
    int i = 5;
    int *p = &i;

    printf("Value that p points to: %d\n", *p);
}
```

11.3.3 L Values

- `&` requires an L value as argument.

```
int i = 5;
int *p = &i; // &5 or &(i+1) is not allowed
```

- `*` is the counterpart of `&`

```
printf("%d\n", *&i); // ==> prints 5
```

- Therefore, `*` returns an L value.

11.3.4 Example 2

```
int main() {
    int i = 6;
    int *p = &i;

    (*p) = 8; // (*p) is an L value!

    printf("Value of i: %d\n", i); // ==> 8!
}
```

11.3.5 What just happened?

```
int *p = &i;
(*p) = 8;
```

- `p` contains `&i`
 - Therefore `*p` is the same as `&i`
 - `&i` is the same as `i`.
- `(*p) = 8` therefore means `i = 8`.

11.3.6 Assignments

- Remember: In C assignments always copy values

```
int i, k;

i = 5;
k = i;
i = 6;

printf("%d\n", k); // What is the output?
```

11.3.7 Assignments with pointers

```
int i, k, *p, *q;

i = 1; k = 2;
p = &i; q = &k;
*p = k;
p = q;
*p = 3;

printf("%d %d\n", i, k); // What is the output?
```

11.3.8 Example (1)

```
i = 1; k = 2;
```

i	1
k	2
p	?
q	?

11.3.9 Example (2)

```
p = &i; q = &k;
```

i	1
k	2
p	address of i
q	address of k

11.3.10 Example (3)

```
*p = k;
```

i	2
k	2
p	address of i
q	address of k

11.3.11 Example (4)

```
p = q;
```

i	2
k	2
p	address of k
q	address of k

11.3.12 Example (5)

```
*p = 3;
```

i	2
k	3
p	address of k
q	address of k

11.4 Stackframe

11.4.1 Variables in memory

- A variable is a labelled memory segment at a certain address
 - Address is assigned by the compiler (for local variables)
- For each procedure call, a new memory segment is created.
 - Called the stackframe.
 - Local variables are stored in this segment.
 - * Stackframe = road name
 - * Exact position inside stackframe = street number

11.4.2 Properties of Stackframes

- Stackframe of a procedure is always stored on top of the stackframe of the calling procedure
 - After a procedure call has ended, the stackframe is removed.
 - Memory segment of former stackframes is reused in later procedure calls.

11.4.3 Example

```
void foo() { int a; printf("address of a: %p\n", &a); }
void bar() { int b; printf("address of b: %p\n", &b); }
int main() { foo(); bar(); }
```

address of a: 0x7fff8e566064

address of b: 0x7fff8e566064

- Output will differ depending on system!
- Stackframe of `bar` reuses stackframe of `foo`, hence the addresses of the local variables in the procedures are identical.

11.5 Call by pointer

11.5.1 Parameters in functions

Parameter in functions are copied!

```
void foo(int i) {
    i = 2;
}
```

```
int main() {
```

```

    int i = 1;
    foo(i);
    printf("%d\n", i); // output?
}

```

11.5.2 Pointer as parameters for procedures

- Pointer variables in parameters can contain addresses of variables outside of the local stackframe!

```

void foo(int *q) {
    *q = 2;
}

int main() {
    int i = 1; int *p = &i;
    foo(p);
    printf("%d\n", i); // output?
}

```

11.5.3 Example (1)

```

>>int i = 1; int *p = &i;
    foo(p);

```

main	i	1
	p	address of i

11.5.4 Example (2)

```

    int i = 1; int *p = &i;
>>foo(p); // void foo(int *q)

```

main	i	1
	p	address of i
foo	q	address of i

11.5.5 Example (3)

```

void foo(int *q) {
>>*q = 2;
}

```

main	i	2
	p	address of i
foo	q	address of i

11.5.6 Side effects and functional programming

- Side effect = A procedure modifies something outside of its stackframe.

```
void foo(int *q) {
    *q = 2;
    // side effect: q points to a variable that is
    // declared outside of foo
}
```

- function = procedur without side effect
 - Advantages of programming without side effects (functional programming)
 - * Functions always return the same value for the same arguments
 - * The function call can be moved without problems

11.5.7 Exercise

- Write a procedure `swap` that uses call by pointer to swap the values in two variables.

```
// Example usage
int i = 12, k = 7;
if(i > k) swap(&i, &k);

// now, i contains the smaller value, k the larger one.
```

11.5.8 Solution

```
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

- Why is `int *t = a; *a = *b; b = t;` wrong?

11.5.9 Pointers as return values

- Returning a pointer from a procedure can be dangerous

```
int *foo() {
    int i = 5; int *p = &i;
    return p;
}

int main() {
    int *q = foo();
    // q points to a variable that does not exist anymore!
}
```

11.5.10 A more complex example

- Yet, returning a pointer allows very powerful procedures

```

int *maxPtr(int *a, int *b) {
    // returns the pointer to the variable with the larger value
    if(*a > *b) return a; else return b;
}

int main() {
    int i = 3, k = 9;
    (*maxPtr(&i, &k)) = 0;

    // the variable that contains the larger value is set to 0

    printf("%d %d\n", i, k);
}

```

11.5.11 Exercise

- Implement a procedure `void sort(int *a, int *b)` that sorts two values:

```

int i = 8, k = 4;

sort(&i, &k);

printf("%d %d\n", i, k);

```

11.6 Arrays and pointers

11.6.1 Pointers to arrays

```

int array[2];

printf("%p %p\n", &array[0], &array[1]);
printf("%p\n", array); // !

```

- In an array, all elements are arranged next to each other.
 - Distance is the amount of bytes required for datatype.
- array itself contains a pointer to the beginning of then array.

11.6.2 Pointer arithmetics

- Pointers are numbers.
 - Limited support for mathematical operations
 - Addition and subtraction are supported.

```

int array[] = {1, 2, 3, 4};

int *p = &array[1]; // p points to array[1] which contains 2.

p = p + 2; // Advance by two integers

printf("%d\n", *p); // output?

```


11.6.3 Stackframe

```
int *p = &array[1];
```

p	address of array[1]
array[0]	1
array[1]	2
array[2]	3
array[3]	4

11.6.4 Stackframe (2)

- `p` is a pointer to an `int`.
- `p + 2` advances by two `ints`.

```
p = p + 2;
```

p	address of array[3]
array[0]	1
array[1]	2
array[2]	3
array[3]	4

11.6.5 Semantics of []

- `array` is a pointer to the first element in an array.
- By adding `n` one obtains a reference to the n th element in the array.

Therefore, the following notations are equivalent:

```
int array[] = {1, 2, 3};

printf("%d\n", array[1]);

printf("%d\n", *(array + 1));
```

- Always prefer `array[1]` over `*(array + 1)`!

11.6.6 Pointer arithmetic vs arrays

11.6.6.1 Sum of int array using array notation

```
int array[] = {1, 2, 3, 4, 5, 6};
int len = 6;

int sum = 0;

for(int i = 0; i < len; ++i) {
    sum += array[i];
}
```

11.6.6.2 Sum of int array using pointers

```
int array[] = {1, 2, 3, 4, 5, 6};
int len = 6;
int *end = array + len; // one past the last element

for(int *p = array; p < end; ++p) {
    sum += *p;
}
```

- Such a variable *p* is also often called an *iterator*

11.6.6.3 Misunderstanding pointer arithmetics

The following code does NOT use pointer arithmetics but array notation!

```
int array[] = {1, 2, 3, 4, 5, 6};
int len = 6;

for(int i = 0; i < len; ++i) {
    sum += *(array + i); // Bad syntax!
}
```

11.7 Function Pointers

11.7.1 Functions as datatypes

- Functions are types
 - Variables can therefore also contain functions
- Many advantages
 - Reuse of codeparts even if some part of the functionality differs
- Functional languages
 - Function is a common datatype in functional programming languages like Lisp, OCaml or Scala.
 - In C using function pointers

11.7.2 Motivation

- Multiply each element of an array by 2

```
int dbl(int i) { return 2 * i; }

void dbl_array(int len, int array[]) {
    for(int i = 0; i < len; ++i) {
        array[i] = inc2(array[i]);
    }
}
```

- Now, let's increment every element of an array

```
int inc(int i) { return i + 1; }

void inc_array(int len, int array[]) {
    for(int i = 0; i < len; ++i) {
        array[i] = inc(array[i]);
    }
}
```

```

    }
}

```

- Violation of *Don't-Repeat-Yourself* principle
- By providing an additional argument that can contain a function, the `sqr_array` and `inc_array` can be combined

```

int inc(int i) { return i + 1; }

int dbl(int i) { return 2 * i; }

void do_array(int len, int array[], int (*fun) (int)) {
    for(int i = 0; i < len; ++i) {
        array[i] = fun(array[i]); // fun is an argument
    }
}

int main() {
    int array[] = { 1, 2, 3, 4 };
    do_array(4, array, inc); // increment
    do_array(4, array, dbl); // increment
}

```

11.7.3 Syntax of function pointers

- General syntax of pointers

```

int *pointer; // * is always next to name.
int (*pointer); // brackets are allowed

```

- Function prototype

```

int function(int);

```

- Return value ahead of name
- Parameters inside () behind name.
- Information that it is a pointer is always an immediate prefix to the name

```

int (*function_pointer)(int);

```

- Brackets are important to distinguish it from a function that returns a pointer!

11.7.4 Exercises

- Try to define function pointers for the following functions

```

int foo(int, int); // prototype of a function

char *strcmp(char* s, char* t); // string compare

```

- Implement a `max`-function that returns the bigger of two arguments. A function pointer should be used to store the comparison operation.

11.7.5 Example: Custom comparison using function pointer

```

int greater_than(int a, int b) { return a > b; }

```

```
int less_than(int a, int b) { return a < b; }

int max(int a, int b, int (*cmp)(int, int)) {
    if((*cmp)(a, b)) return a;
    else return b;
}

printf("%d\n", max(3, 6, less_than));
```